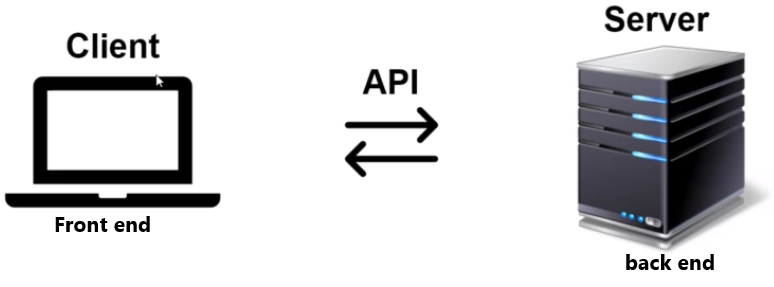
API - application programming interface - it is an interface that allows two applications to talk to each other.





Web = internet

Service = API (it is just a short way of saying API)

Web service = API that uses the internet

So if you put two words together, a web service is just an API that uses the web or the internet. Not all APIs use the internet. Those that don't use the internet are obviously not web services and those that do are web services.

**So all Web Services are API.**

But not all APIs are web services. Not all APIs use the internet.

Web Services use:

1) - XML or JSON to format the data over the internet. When you send **data** over the internet, you have to use a certain format to send it over. You can't just use “notepad” or “word” or something like that.

2) - REST, SOAP or XML/RPC используемые протоколы (имеется ввиду правила зборки HTTP. Eсли это SOAP то это один способ, если REST то другой. Но всё происходит в рамках HTTP)

**HTTP**

When we call an API over the internet we use something that is called **HTTP**. So we need to know what HTTP is, to understand what Web Services are.

HyperText Transfer Protocol.

What is regular text? For example “www.google.com”. What does regular text do? nothing. Hyper text meaning that this text can go somewhere else. (hyper means somewhere else). <https://www.google.com>, this thing enables this to go to the google computer (server).

What parts in HTTP request? : a) start line (that starts the program, tells it what to do.(**mandatory**).

b) headers (that provides additional information other than the start line.

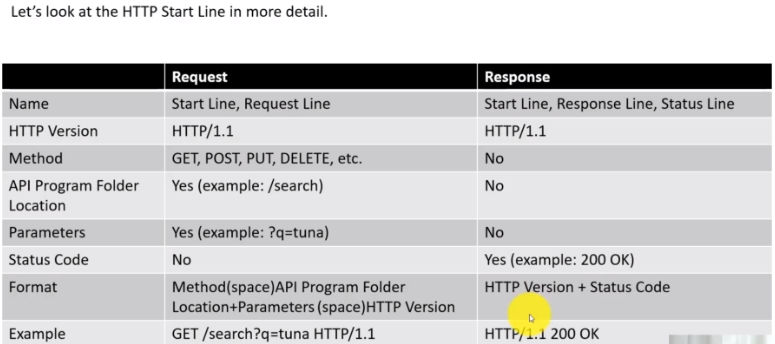
c) blank line

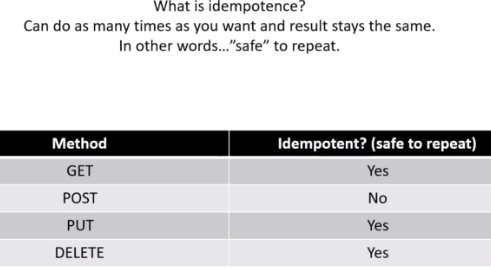
d) body (where we can send information)

What parts in HTTP response? the same 4 parts. What is in this parts:

| HTTP | REQUEST | RESPONSE |
| --- | --- | --- |
| start line | (also called request line)  version,  method(put, get, post, delete),  parameters, folders | (also called response line or status line)  version,  status code |
| headers  (additional info) | For example: 1)-host: [www.google.com](http://www.google.com) <-host is a header. 2) Content-type is also a header that describes the type of content in the body.  2)-token (for example security token) | cookie |
| blank line (used for separating the content) |  |  |
| body | depends of the type of the request | What we requested.  HTML |

**START LINE**



What is idempotence - it means you can repeat something as many times as you want and the results will stay the same, in other words “safe” to repeat. 

**HTTP HEADERS (in other words headers line)**

HTTP headers contain metadata in key-value pairs that are sent along with HTTP requests and responses. They can be used to define caching behavior (поведение кэширования), facilitate authentication, and manage session state. HTTP headers help the API client and server communicate more effectively

For example: Content-Type: text/html; charset=UTF-8 ←Context-Type and charset are headers!!!

Request headers are sent by the client to the server and contain information and instructions related to the requested resource, while response headers are sent by the server to the client and provide metadata, instructions, and additional information about the response itself.

**Some of the most commonly used request headers are:**

**Accept**

The Accept header defines the media types that the client is able to accept from the server. For instance, Accept: application/json, text/html indicates that the client prefers JSON or HTML responses. This information allows the server to send a resource representation that meets the client’s needs.

**User-Agent**

The User-Agent header identifies the web browser or client application that is making the request, which enables the server to tailor its response to the client. For instance, if the User-Agent header indicates that the request is coming from the Chrome browser, the server may include CSS prefixes for CSS properties that are compatible with Chrome.

**Authorization**

The Authorization header is used to send the client’s credentials to the server when the client is attempting to access a protected resource. For instance, the client might include a JSON Web Token (JWT) as the value of the header, which the server will then verify before returning the requested resource.

**Content-Type**

The Content-Type header identifies the media type of the content in the request body. For instance, Content-Type: application/json indicates that the request body contains JSON data. This information helps the server successfully interpret and process the payload (полезная нагрузка).

Cookie

The client can use the Cookie header to send previously stored cookies back to the server. The server then uses these cookies to associate the request with a specific user or session. This header plays an important role in delivering personalized experiences, as it enables the server to remember a user’s login state or language preference.

**The most common response headers are:**

**Content-Type**

The Content-Type response header is the counterpart of the Content-Type request header, as it indicates the type of data that the server is sending to the client. The header value typically includes the media type (such as text/html, application/json, image/jpeg, and audio/mp3), as well as any optional parameters.

**Cache-Control**

The Cache-Control header controls caching behavior in the client’s browser or intermediate caches. It defines how the response can be cached, when it expires, and how it should be revalidated. For example, Cache-Control: max-age=3600, public instructs the client to cache the response for a maximum of 3600 seconds (1 hour) and allows caching by public caches.

**Server**

The Server header includes the name and version of the server software that generated the response, as well as information about the server’s technology stack. For instance, Server: Apache/2.4.10 (Unix) indicates that the response was generated by the Apache web server version 2.4.10. It’s important to note that the Server header is informational and doesn’t affect the API’s functionality.

**Set-Cookie**

The Set-Cookie header instructs the client to store a cookie with the specified name, value, and additional attributes, such as expiration, domain, path, and security flags. The client will then include the cookie in subsequent requests in order to facilitate stateful communication and personalized experiences.

Content-Length

The Content-Length header, which specifies the size of the response body in bytes, can help the client anticipate how much data it is going to receive. This improves performance by allowing the client to plan in advance for more efficient memory allocation and data processing.

**BLANK LINE**

Without a blank line, you wouldn't be able to recognize where the header ends, and where the body begins.

**BODY**

**The body is what contains content.**

It could be an image, html web page, video or data. It just contains content that you want to send to the API if you are doing the request, or the content that comes from API if its response from the server.

Theris header is called **Content-type** that indicates the type of data that we are sending or receiving and there's 2 types of **data (**now we are talking about data not video or image**)**that is used more often: **JSON**, **XML** <- these typical ways to send data to the API or receive it back. What I mean by data is **text information**. Very important to know what JSON and XML are.

What is stateless/stateful

Network Protocols for web browser and servers are categorized into two types: Stateless Protocol, and Stateful protocol.

**Stateless Protocols** are the type of network protocols in which Client send request to the server and server response back according to the current state. It does not require the server to retain (сохранить) session information or a status about each communicating partner for multiple requests.

**HTTP** (Hypertext Transfer Protocol), UDP (User Datagram Protocol), DNS (Domain Name System) are examples of Stateless Protocol.

Salient features (отличительные черты) of Stateless Protocols:

1)-Stateless Protocol simplifies the design of the Server.

2)-The stateless protocol requires less resources because system do not need to keep track of the multiple link communications and the session details.

3)-In Stateless Protocol each information packet travel on it’s own without reference to any other packet.

4)-Each communication in Stateless Protocol is discrete and unrelated to those that precedes or follow.

2 ways of doing HTTP: 1) REST, 2)SOAP, they are stateless by default, because they use HTTP and HTTP is statless by default it means it does not remember what happened between one http call and the other.

For example: if someone call you on the phone in 1999 you dont know who, you would answer every call in the same manner (Hello…).Same here - server response on every request in the same manner.

**Stateful Protocols**, Its not HTTP !!!.

In Stateful Protocol If client send a request to the server then it expects some kind of response, if it does not get any response then it resend the request. FTP (File Transfer Protocol), TCP, and Telnet are examples of Stateful Protocol.

Salient features of Stateful Protocol:

1)-Stateful Protocols provide better performance to the client by keeping track of the connection information.

2)-Stateful Application require Backing storage.

3)-Stateful request are always dependent on the server-side state.

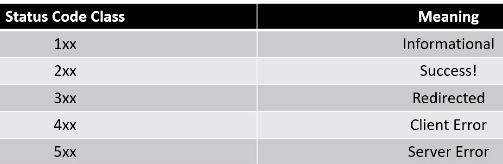
4)-TCP session follow stateful protocol because both systems maintain information about the session itself during its life.

Comparisons between Stateless and Stateful Protocol:

| Stateless | Stateful |
| --- | --- |
| does not require the server to retain the server information or session details. | require the server to save the status and session information. |
| there is no tight dependency between server and client. | there is tight dependency between server and client |
| simplify the server design. | makes the design of server very complex and heavy |
| works better at the time of crash because there is no state that must be restored, a failed server can simply restart after a crash. | does not work better at the time of crash because stateful server have to keep the information of the status and session details of the internal states. |
| handle the transaction very fastly. | handle the transaction very slowly. |
| are easy to implement in Internet. | are logically heavy to implement in Internet. |
| Scaling architecture is relatively easier. | It is difficult and complex to scale architecture. |
| The requests are not dependent on the server side and are self contained. | The requests are always dependent on the server side. |
| To process different information at a time , different servers can be used. | To process every request , the same server must be utilized. |
| Example of Stateless are UDP , DNS , HTTP , etc. | Example of Stateful are FTP , Telnet , etc. |

So then how it is possible that Amazon seems to remember over time the items you put in the cart?

The answer is: cookie .

**STATUS CODE MEANING**

If there is no status code it means the request did not even get to server.

**What is XML** (extensible markup language)

XML holds data.

It holds data that you can send to the API or it holds data that you can receive back from API. The actual **XML is in the body of HTTP.** How is the data contained in XML? Well it`s contained in the tags, just like HTML, the difference is that in HTML the tags themselves are funktions, they are predefined and have some sort of funktionality in XML it means nothing, it just describes the data in it.

The word extensible in the name XML means you can customize these tags to say whatever you want,or add new tags to already existing (и тем самым расширить этот документ) again, in HTML these tags are not extensible (ты не можеш взять и добавить каких то новых тегов которых доселе небыло в HTML) they are predefined.

Example: lets pretend we have an API fo pizza delivery company and they require data from us in the form of XML so that they know what pizza we want, and they tell us xml tags that they want us to full fill for their API.and values for this tags have to be chosen from the predefined.

<pizza>

<size>Medium</size> //here you can put small, medium or large

<topings>

<toping>Cheese</toping> // choose toppings from the menu

<toping>Mashrooms</toping>

</toping>

</pizza>

**What is JSON**

JSON holds data !

For example: Content-Type: application/JSON

XML and JSON are both ways of transferring data using HTTP.

Just like with XML, you put the XML in the body, and if the header Content-Type: application/json, we put JSON in the body.

Example of JSON:

“name”: “The Russian” ← we don't have any tags in JSON

The same example in XML:

<name> The Russian </name>

Now we convert previous example with pizza to JSON:

{ //начала описания объекта начинается/заканчивается с фигурных скобок

“pizza”: [ //значение поля состоит из массива в котором тоже лежит 1 обьект

{ //и так как это объект также открываем фигурную скобку и описываем поля

“size” : “Medium”,

“toppings” : [“cheese”, “mushrooms”]

}

{...} //можно добавить ещё обьектов

] //чтобы открыть такой файл в браузере надо накинуть расширение

} //что - то типа json-wue, редактировать с помощью notepad++

//далее нажать **ctrl+O** и выбрать файл на открывание(или указать путь к файлу)

//открыть можно в любом браузере

Comparison XML and JSON:

xml is more powerful, and on the ather hand it makes it more complicated. JSON is very simple that makes it lightweight and smaller. For popularity XML goes down and JSON is going up. When decide wich wone to use it depends on the API. If API is an XML API then we have to provide it XML, and so on with JSON, sometimes theris API with both.

**SOAP/REST**

They are just ways of doing API.<--непонятно, уже понятно имеется ввиду что это способ организаци API

Web services (API) that go over internet, use HTTP, **SOAP and REST just ways to form that HTTP request and HTTP response.**

**SOAP-**simple object access protocol. And it's not simple, it's just a subjective term.Basically its a way to access a web service (API) by following rules(protokols). Again! We access the API using HTTP!!!

So SOAP are rules (protokols) **to form** the HTTP request and HTTP response.

Every API that uses the SOAP has a **WSDL** (**W**eb **S**ervices **D**escription **L**anguage), this is XML and

It describes what you can send and get to/from SOAP API (the Web Service).

ниже фрагмент **WSDL** файла **из** API где элементом является описание тега, а атрибут name является именем, т.е. то как должен называться тег в xml в запросе клиента



As you remember in HTTP there are 4 parts, according to a SOAP, when we generate HTTP it has : **1)-in a start line** we always have to use:

**1)- method POST**. The reason why is not because SOAP always create information, but because the method is not used, this is just a placeholder(заполнитель).

**2)- WSDL location**

**3)-HTTP version**

Example: Start Line Post WSDL HTTP version

**2)- in a headers line** we have to have header Content-Type text/xml. In another words we always need to have XML in the body.Because SOAP works only with XML.

**3)**-**blank line** - nothing, its just for separating HTTP headers from the body.

**4)**-**body** - as we already know from header line, we have to have XML included in it. (probably SOAP use only XML). And XML formed using the web services (API) description from **WSDL.**

Every time we are creating a XML in the body of HTTP using SOAP, we have to include something that is called **soap Envelope** wich contains:Namespaces, xsd(пока не понятно что это), тэги или по другому элементы(название этих тегов описано в WSDL).

-**Namespaces**. XML Namespaces provide a method to avoid element name conflicts. In XML, element names (by saying element names means tag names) are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications.

This XML carries **HTML** table information:

<table>

<tr>

<td>Apples</td>

<td>Bananas</td>

</tr>

</table>

This XML carries **information about a table** (a piece of furniture):

<table>

<name>African Coffee Table</name>

<width>80</width>

<length>120</length>

</table>

If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning. A user or an XML application will not know how to handle these differences.

Name conflicts in XML can easily be avoided using a name prefix. This XML carries information about an HTML table, and a piece of furniture:

<h:table>

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

<f:table>

<f:name>African Coffee Table</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

In the example above, there will be no conflict because the two <table> elements have different names because of the prefix.

When using prefixes in XML, a **namespace** for the prefix must be defined.

The namespace can be defined by an **xmlns** attribute in the start tag of an element.(n.s.- means namespace)

The namespace declaration has the following syntax. **xmlns:prefix="URI"**.интересно а префиксы можно использовать какие хочеш?

<root>

<h:table xmlns:h="<http://www.w3.org/TR/html4/>"> //**xmlns** attribute in the start tag of an element.

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

<f:table xmlns:f="https://www.w3schools.com/furniture">

<f:name>African Coffee Table</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

</root>

The xmlns attribute in the first <table> element gives the h: prefix a qualified namespace.

The xmlns attribute in the second <table> element gives the f: prefix a qualified namespace.

Namespaces can also be declared in the XML root element:

<root xmlns:h="<http://www.w3.org/TR/html4/>" //declaration of namespace in root

xmlns:f="https://www.w3schools.com/furniture">

<h:table>

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

<f:table>

<f:name>African Coffee Table</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

</root>

Note: The namespace URI is not used by the parser to look up information.

The purpose of using an URI is to give the namespace a unique name.

**!!!** Below is example of SOAP body, that contains mandatory element <soap:Envelope> which contains…



SOAP was created the same organization that created XML (they use XML for the WSDL and they use XML in the body)

В ОБЩЕМ, что я понял, когда API является SOAP сервисом, то у него есть WSDL - описание всего того что умеет эта API и как к ней формировать запросы и в какой форме приходят риспонсы. Тестировать SOAP можно в Postman или SOAP UI. Идея в том чтобы используя WSDL правильно (корректно) сформулировать риквест и получив риспонс от API, сверить его с документацией на соответствие.

**REST (representational state transfer)**

It's a modern **way** of doing API. In another words REST is architectural design.

The difference between SOAP and REST is that SOAP always uses the POST method in a start line and it's never changed. REST uses different types of HTTP methods (POST, GET, PUT, DELETE) in the start line to tell the server what we need. In the body you can use:XML, JSON, PNG, JPEG, VIDEO…any, to say not only XML as in SOAP.

NOTE! - REST does not has a methods: post, get, delete… it is all the methods of HTTP.

REST API - (Representational State Transfer) архитектурный стиль взаимодействия компонентов распределённого

приложения в сети. Т.Е. это не какая то технология это просто подход, это архитектурный стиль который диктует определённые правила, чтобы одни программы могли по определённым правилам общаться с другими программами, Опять же - это набор правил с помощью которых мы можем выстроить это взаимодействие.

ТРЕБОВАНИЯ К АРХИТЕКТУРЕ REST

**1**) - Клиент - серверное взаимодействие. Важно понимать, что сервер сам по себе никогда никакие запросы посылать никуда не будет. Ключивой момент - сервер отправляет ответ только когда клиент делает запрос. Пример: просматриваем страницу в youTube там определённое количество “лайков” оно не меняется динамически, чтоб увидеть изменения необходимо обновить страницу, а обновление это и есть запрос, по которому и придёт ответ - таже страница с обновлёнными данными (в том числе и с другим числом лайков).

**2**) - Операции над ресурсами. Ресурсами выступает например у нас есть интернет магазин и продукты всевозможные в нём: телефоны, телевизоры и т.д. это и есть ресурсы и мы можем их покупать, ложить в корзину, бронировать, совершать над ними какието действия эти действия и есть операции . Ещо к примеру YouTube, видео на этом сайте это и есть ресурсы которыми можно пользоватся - просматривать , а это и есть операции..

3) - Передача данных с помощью HTTP. HTTP - это основа любого API,

4) - Отсутствие состояния (STATELESS). Например я пользуюсь YouTube и просматривая контент сервер предлагает мне схожие тематически видео, вот этот момент и есть состояние (т.е. мои интересы это и есть состояние), но сам сервер не хранит информацию о клиентских предпочтениях (состоянии), это очень много инф., вместо этого состояние хранится на машине клиента, это называется COOCKIES, но сам принцип видимо по отношению к серверу называется STATELESS

5) - Форматы передачи данных:JSON, XML, любой тип данных можно передать с помощью REST, это не SOAP, где в body есть только XML.

**WHAT IS CACHE?**

A cache is where something is kept in memory. For example when you request something from the server. and gat the result you can keep it in your cache in case if some time afterwards you again makes the same request you don't need to go to the server you can get that response from the cache. The reason why it has to do with REST is because of get function (GET method). If you have the result in your cache, you don't need to go to the server again.

**Cloud computing**

Облачные вычисления - это доставка IT ресурсов через интернет по запросу с оплатой по факту использования. Т.е. - это как арендовать вычислительные мощности, не покупать а брать на сьём. Пример: имеется идея - уникальная интернет площадка для продажи и обмена раритетных книг. Предположим что для реализации этого веб приложения уже написан код, остаётся выложить в интернет на сервер, на  **традиционной инфраструктуре понадобится :**

1)Физические сервера для баз данных, выб и т.д.

2)Физическое место для данного сервера, т.е. что-то типа data center.

3)Охлаждение воздуха, кондиционеры.

4) Доступ в интернет, очень скоростной, так-как будет много пользователей.

5)Электричество

6)Обеспечение безопасности (охрана).

7)Работники для обслуживания оборудования.

8)Время на приобретение, настройку и отладку серверов…

9)Если идея оказалась успешной необходимо масштабировать data center, т.е. открывать, в зависимости от рынка ещё такие же центры.

10)Если идея на каком то этапе провалилась то потребуются ресурсы на ликвидацию всего оборудования.

**облачная инфраструктура:**

1)Любой комп с любым браузером.

2)Доступ в интернет

3)Время для настройки

… и всё. Не надо угадывать сколько серверов потребуется.

**Ёмкость серверов vs используемость серверов.**

**Традиционная инфраструктура**

По началу открытия физического data center количество купленных серверов, как правило, не используется на 100%, скорее всего это будет какой то небольшой процент так как ваша идея ещо не набрала популярность. С течением времени идея становится более популярной и на каком-то этапе наступает надобность приобрести ещё какое-то (а сколько взять чтоб не лохануться) количество серверов, что связано с определенного рода трудностями и энергозатратами в лучшем случае, а ведь бывает и такое что серверов за много и они устаревают, или наобоот их недостаточно и приложение не справляется с потоком пользователе и вы теряете клиентов. И вот этот момент при прочих равных он наступает периодически и требует организаторского решения, сколько бы серверов не было-бы запущено.

**облачная инфраструктура:** позволяет достичь эластичности в использовании серверов просто за счет настроек. Чем больше запросов на ваши ресурсы тем больше подключаются облачные сервера предоставляя необходимую мощность и наоборот если спроса мало то и сервера отключаются и не тратят твои деньги. Ты и не переплачиваеш за лишнее и в то же время всем всего хватает.

**Отсюда и определение, ты всегда платишь за ресурсы которые фактически используешь и никогда не платишь дополнительно.**

**Плюсы использования cloud computong:**

1)- Гибкость, от идеи до реализации за считанные минуты.

2)-Эластичность - автоматическое выделение ресурсов которые вы фактически используете.

3)-Экономия затрат.

4)-Глобальность - мультирегиональная инфраструктура за минуты. Надо ещё серверов - пожалуйста, много и надо уменьшить - пожалуйста.

**Лидеры в предоставлении данных услуг:**

1)Amazon web services (AWS)

2)Microsoft (AZURE)

3) Google cloud

Я тут перечитал через пару месяцев и хочу немного обобщить: SOAP and REST is just diferent ways of doing http.JSON - тип данных.